Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms

Tchimou N'Takpé, Frédéric Suter

Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms*

Tchimou N'Takpé Frédéric Suter
UHP Nancy 1 / LORIA†
Campus Scientifique - BP 239
F-54506 Vandoeuvre-lès-Nancy Cedex
{Tchimou.Ntakpe, Frederic.Suter}@loria.fr

Abstract

While most parallel task graphs scheduling research has been done in the context of single homogeneous clusters, heterogeneous platforms have become prevalent and are extremely attractive for deploying applications at unprecedented scales. In this paper we address the need for scheduling techniques for parallel task applications for heterogeneous clusters of clusters by proposing a method to adapt existing parallel task graphs scheduling heuristics that have proved to be efficient on homogeneous environments. The contributions of this paper are: (i) a novel "virtual" cluster methodology for handling platform heterogeneity; (ii) a novel task placement step, designed to determine whether the placement step of heuristics for homogeneous platforms is adapted to the heterogeneous case; (iii) an empirical evaluation in a wide range of platform and application scenarios. This study shows that the proposed heuristics achieve better performance than the original when platform are heterogeneous and we discuss a number of trends apparent in our results.

1 Introduction

To face the increasing computation and memory demands of parallel scientific applications, a recent approach has been to aggregate multiple homogeneous compute clusters either within or across institutions. Typically, clusters of various sizes are used, and different clusters contain nodes with different capabilities depending on the technology available at the time each cluster was assembled. Therefore, the computing environment is at the same time attractive because of the large computing power, and challenging because of its heterogeneous nature.

One way to exploit this computing power is to maximize the degree of parallelism of a given application by combining two kinds of parallelism, *data* and *task* parallelism. The former consists in applying the same operation in parallel on different elements of a data set, while the latter corresponds to concurrent computations on different data sets. Several data-parallel computations can then be executed concurrently in a task-parallel way. In this context scheduling a parallel application consists in scheduling a *parallel tasks graph*. Informally a parallel task is a computation made of elementary operations that exhibits enough parallelism to be executed efficiently on more than one processor. In this paper, we only consider a sub-class of parallel tasks: the *moldable tasks* [10]. The number of processors on which a moldable task is to be executed is not fixed *a priori* but determined before the execution. However, this number of processors can not be modified once the task is started. Then we consider task-parallel applications described as a graph in which each task (*i.e.,* node) is itself moldable.

The problem of scheduling applications represented by such graphs is to find the number of processors and the cluster on which to execute each task of the graph while satisfying resource constraints and task dependencies. The objectives are to minimize the completion time of the application while keeping a "good" balance between execution time and resource consumption.

Most parallel task graphs scheduling research has been done in the context of homogeneous platforms [6, 7, 8, 9]. However, as explained above, heterogeneous platforms have become prevalent and are extremely attractive for deploying applications at unprecedented scales. Consequently, there is a need for scheduling techniques for parallel task applications in the heterogeneous case. Two approaches can be considered. The first is to take an existing sequential task scheduling algorithm for heterogeneous platforms and to develop a new algorithms for parallel task scheduling on heterogeneous platforms. An heuristic fol-

*This work is partially supported by the ARC INRIA OTaPHe, the Conseil Régional de Lorraine and the Government of Côte d'Ivoire.

†UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1.

Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)
0-7695-2612-8/06 $20.00 © 2006 IEEE

lowing this first approach has been proposed in [1]. This paper focuses on a second approach: modifying existing parallel task graphs scheduling heuristics [7, 8] designed for homogeneous platforms in order to handle resource heterogeneity. We constrain the allocation of a given parallel task onto processors of a single cluster in order to remain effective and be able to use models designed for homogeneous platforms. The contributions of this paper are: (i) a novel "virtual" cluster methodology for handling platform heterogeneity; (ii) a novel task placement step based on the *sufferage* idea [2], designed to determine whether the placement step of heuristics for homogeneous platforms is adapted to the heterogeneous case; (iii) an empirical evaluation in a wide range of platform and application scenarios.

This paper is organized as follows. Section 2 presents the platform and application models used in this paper. Section 3 discusses related work and formalizes the problem we address. Section 4 presents how we adapt the CPA algorithm to target heterogeneous platforms. Section 5 presents our evaluation methodology and Section 6 presents our evaluation results. Section 7 concludes the paper with a summary of our contributions and future work.

2 Platform and Application Models

Our heterogeneous platform model consists of a cluster of clusters, *i.e.,* a heterogeneous collection of homogeneous clusters connected through a high bandwidth backbone as shown in Figure 1. Such platforms are representative of a certain class of currently available Grid platforms as they typically consist of clusters located at different institutions, and institutions typically build homogeneous clusters. Hence we have C clusters, where each cluster C_i, for $i = 1, \ldots, C$, contains P_i identical processors for a total of P processors in the platform. Processor speed and LAN bandwidth within clusters are not necessarily the same between two different clusters. Each processor executes tasks one after the other in a space-sharing model.

Inside each cluster processors are connected by a switch and can access to the backbone through a gateway that limits the number of processors that can simultaneously send data over the backbone. Figure 1 shows our platform model and the fixed TCP routes connecting two processors either in two separate clusters (processors A and B) or within the same cluster (processors a and b).

A parallel application can be modeled by a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{t_i : i = 1, \cdots, N\}$ is a set of N nodes (or tasks) and $\mathcal{E} = \{E_{i,j} : i, j = 1, \cdots, N\}$ is a set of E edges. Each task has a computation cost, *e.g.,* a number of flops, which leads to different computation times on different processors. An edge in the DAG corresponds to a task dependency (communication and precedence constraint.) To each edge $E_{i,j}$ is associated D_{ij}, the amount of data in bytes that task t_i sends to task t_j. Each such transfer incurs a communication cost that depends on network capabilities. It is assumed that if two tasks are assigned to the same processor set there is no communication cost but a just a task dependency. Moreover we assume that several communications may be performed at the same time, possibly leading to network contention.

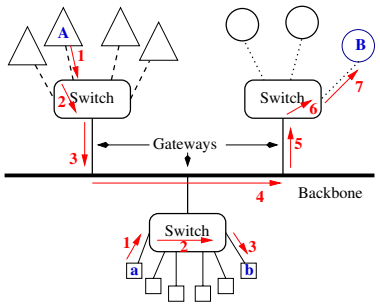

Figure 1. Platform model with routes between two processors A and B located in two separate clusters and between two processors a and b of the same cluster.

A task without any input edge is called an *entry* task while a task with no output edge is called an *exit* task. A task is said *ready* when all its predecessors have completed.

In a homogeneous processor system, the execution time of a parallel computation can be expressed by classical speed-up models. In this paper we use the Amdahl's law to model the parallel execution of a given task t_i:

$$T_\omega(t_i, N_p(t_i)) = \left(\alpha + \frac{1-\alpha}{N_p(t_i)}\right) \cdot T_\omega(t_i, 1)$$

where $N_p(t_i)$ is the number of processors allocated to task t_i, $T_\omega(t_i, 1)$ is the execution time of this task on a single processor of this cluster C_i and α the part of the task that cannot be parallelized. $T_\omega(t_i, N_p(t_i))$ then denotes the execution time of a particular task on a given number of processors.

We assume that a parallel task only uses processors within a single cluster. Deploying a task on several geographically distant clusters would lead to an important overhead. Furthermore a convenient way to ensure that we can use that model is to always map parallel tasks within a single cluster.

In the case of sequential tasks an inter-task communication simply implies a point-to-point data transfer. But in the case of parallel tasks it involves a potentially complex data redistribution, since source and destination data items are themselves distributed among multiple processors.

Finally we define $T_b(t_i)$ as the *bottom level* of a task t_i, *i.e.,* the cost of the longest path, in terms of execution time, from t_i, including it, to any exit task.

3 Related Work

The task scheduling problem being NP-hard in the strong sense even for sequential tasks and when an infinite number of processors are available, several heuristics have been designed to schedule parallel tasks. In [4] an approximation algorithm for scheduling a set of tree precedence constrained moldable tasks for the minimization of the parallel execution time on several multi-processors is described. In [1], the authors proposed one of the few parallel task scheduling heuristics specifically designed for heterogeneous platforms. They adapted an existing scheduling algorithm for task-parallel applications on heterogeneous platforms to the case of applications consisting of a DAG of parallel tasks. In that paper a limitation of sequential task scheduling heuristics designed for heterogeneous platforms is also shown as the maximal number of processors that can be used simultaneously is determined by the width of the task graph. But when considering parallel tasks, more of the available resources can be used. In the present work we take the complementary approach: adapting an existing algorithm for such applications on homogeneous platforms, so that it is applicable to heterogeneous platforms.

3.1 Scheduling on homogeneous platforms

If some theoretical work exist [6], most of parallel task scheduling algorithms [7, 8, 9] proceed in two steps and have been designed to target homogeneous platforms. Their first step aims at finding an optimal *allocation* for each task, that is the number of processors on which the execution time of a task is minimal. Depending on the parallel task model, a minimal execution does not necessarily imply the largest number of processors as the cost function may not be monotonous. Their second step determines a *placement* for the allocated tasks, that is the actual processor set where execute each task that minimizes the total completion time of the application.

In [8], authors extract DAGs from C, Fortran or Matlab codes on which is applied their scheduling algorithm, TSAS (Two Step Allocation and Scheduling). This heuristic is based on convex programming, allowed by posynomial properties of chosen cost models, and some properties of their structure. Authors of [9] limit their study to graphs built by serial and/or parallel compositions. Tasks are allocated either in the whole set of processors or an optimal number of subsets, determined by a greedy algorithm. The optimality criterion is the reduction of completion time.

3.2 CPA

For the work described in this paper we focused on [7] which describes a parallel task scheduling algorithms for homogeneous environments named CPA (*Critical Path and Area-based scheduling*). This algorithms aims at finding the best compromise between the length of the *critical path*, *i.e.,* the path in the application task graph on which the sum of the edge and node weights is maximal, and the *average area* which measures the mean processor-time area required by the application. We chose CPA as the start point of our work because it is the most efficient of the different two-steps algorithms cited above. In their allocation procedure Rădulescu *et al.* consider that the execution time of an application can be approximated by $T_p^e = \max\{T_{CP}, T_A\}$, where T_{CP} is the execution time of the application critical path and T_A the average area of the application, defined as:

$$T_{CP} = \max_{t_i \in \mathcal{N}} T_b(t_i), \text{ and} \quad (1)$$

$$T_A = \frac{1}{P} \sum_{i=0}^{N} \left(T_\omega \left(t_i, N_p(t_i) \right) \times N_p(t_i) \right). \quad (2)$$

The goal of CPA is to minimize T_p^e during the allocation step. Knowing that T_{CP} decreases whereas T_A increases when more processors are allocated to a task, the initial allocation leads to a maximal value for T_{CP}. Only one processor is then allocated to each task. Then each iteration allocates one more processor to the most critical task while $T_{CP} > T_A$. This selected task is the task belonging to the critical path that benefits the most of the addition of a processor, *i.e.,* the task t_i for which the ratio $T_\omega(t_i, N_p(t_i))/N_p(t_i)$ decreases the most significantly when $N_p(t_i)$ is incremented.

When verified, the stopping condition ($T_{CP} \leq T_A$) implies that T_p^e will be very close to its minimal value ($T_p^e \approx T_{CP} \approx T_A$) after a good placement.

During the placement step, the ready task with the highest bottom level is considered at each iteration. This step includes data redistribution costs to determine the start date ($T_s(t_i)$) and the end date ($T_f(t_i)$) of each scheduled task t_i.

The time complexity of CPA is dominated by the allocation step and is $O(N(N+E)P)$ in the worst case.

4 Adapting CPA to Heterogeneous Platforms

Our heterogeneous target platform comprising C homogeneous clusters, our idea is to determine for each task a *reference allocation* standing for C potential allocations, one for each cluster. The allocation step will thus consist in determining this reference allocation and designing the function to deduce the number of processors to allocate to a task on each cluster. In the placement step we will select for each task the allocation minimizing its completion time.

As we now have many potential allocations for a given task, we have to redefine formally the execution time of the critical path (T_{CP}) and the average area (T_A) given by equation 2, as they define the stopping condition of the allocation

step of CPA ($T_{CP} \leq T_A$). To do so we introduce the concept of a *reference cluster*. This virtual homogeneous cluster has a cumulated computing power equivalent to that of the whole heterogeneous platform. Its processors have the same speed as those of the slowest processors of the real platform. The number of reference processors composing that virtual cluster is then: $P_{ref} = \left\lceil \sum_{i=0}^{C-1} \frac{P_i}{r_i} \right\rceil$, where r_i is the ratio between the speed of a reference processor and that of a processor of C_i.

Using such a virtual homogeneous platform during the allocation step allows us to keep a low complexity as we have only one allocation to determine for each task and not C. We denote as $N_p^{ref}(t_i)$ the *reference allocation* for task t_i, *i.e.,* the number of processors allocated on the reference cluster. Hence we can translate this virtual processors of the reference cluster into real processors to determine the allocation of each t_i on any given cluster C_j so that the resulting execution is very close to $T_\omega^{ref}\left(t_i, N_p^{ref}(t_i)\right)$, *i.e.,* the execution time of the task on the reference cluster. The execution time of the application critical path can then be redefined as:

$$T_{CP} = \max_{t_i \in \mathcal{N}} T_b^{ref}(t_i), \tag{3}$$

where $T_b^{ref}(t_i)$ is the bottom level of t_i using the reference allocations for the other tasks. The average area is now:

$$T_A = \frac{1}{P_{ref}} \sum_{i=0}^{N} \left(T_\omega^{ref}\left(t_i, N_p^{ref}(t_i)\right) \times N_p^{ref}(t_i) \right). \tag{4}$$

In the next two sections we describe the two steps of our algorithms according to these new definitions.

4.1 Task Allocation

In this first step, we do not take inter-task data redistribution costs into account. As the processor set on which a task will execute is not yet determined, including these costs requires consideration of all possible combinations of source and destination processor sets, which dramatically increases the computational complexity of the allocation step. The data redistribution costs will be taken into account in the second step to determine the start and end date of each task.

At each iteration of the allocation procedure, one supplementary processor will be allocated to a task depending on how much this task will benefit of this new allocation and how critical this task is with regard to the completion of the application. It should be noted that assigning processors to a task may cause its execution time to drop enough for the task to no longer be on the critical path.

To determine the number of processors to allocate to a task on a given cluster from the reference allocation we use Amdahl's law. The following equality:

$$T_\omega^i(t_j, N_p^i(t_j)) \quad = \quad T_\omega^{ref}(t_j, N_p^{ref}(t_j))$$

leads to:

$$f(N_p^{ref}(t_j), t_j, i) = \frac{(1-\alpha) \cdot T_\omega^i(t_j, 1)}{T_\omega^{ref}(t_j, N_p^{ref}(t_j)) - \alpha \cdot T_\omega^i(t_j, 1))}$$

$f(N_p^{ref}(t_j), t_j, i)$ will allow us to determine the number of processor to allocate on each cluster C_i to each task t_j of the real platform depending on its reference allocation:

$$N_p^i(t_j) = \min\{P_i, \lceil f(N_p^{ref}(t_j), t_j, i) \rceil\} \tag{5}$$

To avoid an infinite loop in this procedure, we introduce the concept of *saturated critical path*. The application critical path will be saturated if the reference allocations of its tasks are such that it is not possible to add a processor to any of them, or:

$$\forall t \in \text{critical path}, \nexists i \text{ s.t. } P_i > \lceil N_p^i(t) \rceil. \tag{6}$$

In such a case, the number of processors allocated to any task of the critical path on each cluster C_i is the total number of processors of that cluster. Therefore we cannot further reduce the execution times of these tasks. T_{CP} is then minimal and we can stop the allocation step.

Algorithm 1 is the modified heterogeneous version of the allocation procedure of CPA. Lines 1 to 6 set the initial allocations of each tasks at one processor on each cluster. Line 7 shows the two stopping conditions of this procedure mentioned above. Line 8 and 9 select the task of the critical path that benefits the most of the addition of a processor to its reference allocation. At line 10 we modify the allocation of that task while at line 11 we recompute the bottom level of the different tasks affected by this modification.

Algorithm 1 Processor Allocation

1: **for all** $t_i \in \mathcal{N}$ **do**
2: $\quad N_p^{ref}(t_i) \leftarrow 1$;
3: $\quad N_p^j(t_i) \leftarrow 1, \forall j \in [0, C-1]$;
4: **end for**
5: **while** $T_{CP} > T_A$ and not-saturated critical path **do**
6: $\quad t_i \leftarrow$ critical path task s.t.
7: $\quad (\exists j$ s.t. $\left\lceil f(N_p^{ref}(t_i), t_i, j) \right\rceil < P_j)$ and
8: $\quad \left(\frac{T_\omega^{ref}(t_i, N_p^{ref}(t_i))}{N_p^{ref}(t_i)} - \frac{T_\omega^{ref}(t_i, N_p^{ref}(t_i)+1)}{N_p^{ref}(t_i)+1} \right)$ is maximum;
9: $\quad N_p^{ref}(t_i) \leftarrow N_p^{ref}(t_i) + 1$;
10: $\quad$ **update** T_b^{ref};
11: **end while**

Let $K = \max_{(i,t)} \left\lceil f^{-1}(P_i, t, i) \right\rceil$ be the maximal number of processors that we could allocate to a task on the reference cluster. In the worst case we may perform K iterations of the allocation procedure for each task, leading to a total of $K \times N$ iterations. The inner loop complexity (critical path determination, computation of T_{CP}, T_A, and each task bottom level) is of $O((N+E)\,C)$. The complexity of the allocation step is then of $O\left(N\,(N+E)\,C\,K\right)$ in the worst case.

4.2 Task Placement

The placement step now consists in giving to each ready task t_i its allocation with the earliest finish time. Remind that the reference allocation of a task stands for C potential allocations where C is the number of clusters.

As said before, data redistribution costs are taken into account during this step. We denote as $T_r^k(t_i, t_j)$ the time needed to redistribute data from a task t_i already scheduled on a processor set and a task t_j potentially allocated on cluster C_k. This data redistribution time depends on several factors such as network features, data amount and the number of source and destination processors.

We define $T_m^j(t_i)$ as the arrival date of the last data needed by a task t_i to be executed on cluster C_j:

$$T_m^j(t_i) = \max_{t_k \in Pred(t_i)} (T_f(t_k) + T_r^j(t_k, t_i)), \qquad (7)$$

where $Pred(t_i)$ is the set of the predecessors of t_i. The date at which a task t_i can actually start on cluster C_j is then:

$$T_s^j(t_i) = \max\{avail(N_p^j(t_i)), T_m^j(t_i)\} \qquad (8)$$

where $avail(N_p^j(t_i))$ is the date at which cluster C_j will have $N_p^j(t_i)$ available processors.

The allocation step having produced several potential allocations for each node of the task graph, we studied two different policy to place tasks on clusters. In the next section we adapt the list scheduling algorithm of CPA to the case of multiple candidate allocations. However we keep the same priority function, the bottom level, and placement criterion, the earliest finish time, as in CPA. In section 4.2.2 we propose an alternative approach in which the task selection chooses the task that will suffer the most of not been scheduled on its best allocation.

4.2.1 Bottom Level Based Placement

The bottom level is the priority criterion used in CPA. Among the ready tasks, the task to be considered is the one with the highest bottom level. Once a task has been selected for allocation, the potential allocation that achieves the earliest finish time is chosen. The end date of a task t_i is then:

$$T_f(t_i) = \min_j (T_s^j(t_i) + T_\omega^j(t_i, N_p^j(t_i))) \qquad (9)$$

From this end date, we deduce the cluster, C_j, on which the task will be scheduled, and its start date: $T_s(t_i) = T_s^j(t_i)$.

The combination of the allocation procedure and of this placement algorithm leads to the HCPA (Heterogeneous Critical Path and Area-based) heuristic. The complexity of this placement step breaks down into three components: (i) $O(N + E)$ to compute bottom levels, (ii) $O(N \log N)$ to sort the tasks, and (c) $O(N\,C)$ to schedule tasks to processors (as we try the C candidate allocations, resulting in a total time complexity of $O(E + N \log N + N\,C)$ which can be neglected with regard to the allocation complexity.

4.2.2 Sufferage-Based Placement

The principle of the *sufferage* heuristic [2] is to consider the task that will suffer the most if allocated on its second best allocation, in terms of completion time, and not on its best allocation. To determine what are the best and second best allocations of a task t, we consider the finish time of t with regard to the cluster C_i on which this task is executed as defined by Equation 9. In a second step, we compute the difference between the two best end dates to select which task will suffer the most.

The combination of the allocation procedure and of this placement algorithm leads to S-HCPA (Sufferage-based Heterogeneous Critical Path and Area-based). In the worst case for S-HCPA, we consider $N - i$ ready tasks during the i^{th} iteration of the placement step. Before scheduling the selected task, we estimate the execution time of that task on its C potential allocations. It leads to a worst time complexity of $O(C\,N^2)$ for this step.

The total worst time complexity of HCPA and S-HCPA is then $O(N\,(N + E)\,C\,K)$. Note that in the case of a homogeneous cluster ($C = 1, K = P$) the complexity of our heuristic is the same as the complexity of CPA.

5 Evaluation Methodology

We resort to simulation for evaluating our approach as it allows us to perform a statistically significant number of experiments and makes it possible to explore a wide range of platform configurations. We use the SIMGRID toolkit [5] as the foundation of our simulator. SIMGRID provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms.

We consider platforms that consist of 1, 2, 4, or 8 clusters. Each cluster contains a random number of processors between 16 and 128. We keep the network characteristics fixed and vary the processor speeds to experiment with a range of platform communication/computation ratios.

The links connecting hosts to switches can be Fast Ethernet (bandwidth $= 100Mb/s$ and latency $= 100\mu sec$) or Giga Ethernet (bandwidth $= 1Gb/s$ and latency $= 100\mu sec$) and suffer of contention. Switches have the same bandwidth and latency properties but there is no contention on these links. Gateways have a bandwidth of $1Gb/s$ and a latency of $100\mu sec$ and the backbone connecting the clusters has a bandwidth of $2.5Gb/s$ and a latency of $50msec$.

In our platforms, half of the clusters are connected by Fast Ethernet devices, the other half are connected through Giga Ethernet devices.

Processor speed (in GFlop/sec) is homogeneous within each cluster and sampled from a uniform probability distribution, for various minimum relative speeds (0.25, 0.5, 0.75 and 1) and heterogeneity factors (1, 2 and 5). The upper bound for processor speeds is defined as the product between the minimum speed and the heterogeneity factor. For instance a minimum speed of 0.25 and a heterogeneity factor of 5 mean that every processor of the platform will have a speed comprised between 0.25 and 2.5 Gflops.

The combination of these parameters allows us to generate 40 different platform configurations. As each instantiation has random elements (number of processors per cluster, and processor speeds), we generate five samples for each configuration, for a total of 200 different platforms.

Our random DAGs can be made of 10, 20 or 50 computational nodes. To determine the cost of each of these nodes, we first pick N the data size that will be handled by the task between $4Kb$ and $121Kb$. The rationale behind this upper bound is that we assumed a memory capacity of 1GB per node. With double precision elements, the largest data on which we can compute is of that size. Then we assign a complexity to each node: $a \cdot N$, $a \cdot NlogN$, $a \cdot N^{3/2}$, where a is a factor randomly picked between 2^6 and 2^9. All tasks of a DAG can have the same complexity or we can pick a different complexity for each task. This allow us to modify the communication to computation ratio. Finally we assign to each task its non-parallel part, *i.e.,* the α parameter of the Amdahl's law, with a value between 0 and 0.2. This way we have totally parallel tasks and some tasks for which 20% of their execution has to be done serially. The cost of each transfer node is equal to the data amount N handled by the computational node that produces this transfer.

Four parameters allow us to generate DAGs of different shapes. First we can vary the width of our DAGs. A small value (0.1 or 0.2) induces the generation of thin graphs, *e.g.,* chains, whereas a larger value (0.8) leads to more compact graphs, *e.g.,* fork-joins. This parameter acts on the degree of task parallelism exhibited by a given DAG. We can also change the regularity of the distribution of tasks between the different levels of the DAG (0.1, 0.2 or 0.8). Then the density, *i.e.,* how many dependencies exist between two levels of a DAG, can be tuned. A small value (0.1 or 0.2) will produce a DAG with a few transfer nodes whereas a larger one (0.8) leads to heavy collective communication steps between computations. Finally we can introduce jumps in our DAGs allowing us to generate DAGs with execution paths of different lengths (1, 2 or 4). The combination of these parameters generates 432 different DAG configurations. We generate three samples of each to handle random elements for a total of 1296 DAGs. Even if it is recognized that random graphs do not refer to a real class of applications, the diversity they provide seems sufficient for this experiment.

For all platform and application configurations described above we compare the effectiveness of 5 scheduling algorithms. The first three are CPA, HCPA and S-HCPA that were described in Sections 3 and 4. We also use a fourth algorithm, SEQ, which schedules recursively the nodes of the DAG on the fastest processor of the platform. This algorithm will allow us to compare the efficiency of each heuristic as explained in the next section. Finally we compare our scheduling heuristic to M-HEFT [1] as this heuristic follows a totally different approach to allocate processors to a task. While the CPA-like heuristics aims at reducing the critical path, M-HEFT has a more local point of view and will reduce the completion time of each task without taking the rest of the application into account. With these five heuristic to compare, we obtain a total of 1,296,000 simulation runs for our test plan.

It should be noticed that CPA can also be used to schedule DAGs on heterogeneous platforms even though it was not originally designed for that purpose. To do so, we considered a homogeneous platform for the scheduling process of CPA. This platform has the same number of processors as the heterogeneous one and the speed of each processor is equal to the average of the processor speeds of the heterogeneous platform. But we return to an heterogeneous platform for the simulation. This implies that CPA is allowed to map a task on more than one cluster. Due to our model of parallel tasks based on Amdahl's law that ignores the overhead due to the intra-task communication, the impact on performance will be less than in real life.

To compare the heuristics cited above we decide to use the following metrics. First we consider the makespan, or completion time, of an application defined as the difference between its start and finish dates. Parallel task scheduling heuristics usually aim at minimizing this makespan. As sometimes they also aim at balancing makespan and resource consumption, as in CPA, we study the product between the makespan (in sec.) and the maximal amount of processing power (in GFlop/sec.) used during the schedule. For instance if at most two tasks are executed concurrently during the schedule, the first on 32 processors of speed 1.6 and the other on 24 processors of speed 2.8, we multiply the makespan by $32 \times 1.6 + 24 \times 2.8 = 121.6$. This allows us to see if the gain on the makespan achieved by a given heuristic is not only due to a massive resource usage. But this maximal value can be misleading when only one task is placed on a large set of processors. So we also study the ratio between the makespan and the average processing power used by any task of the application. Another interesting metric is the parallel speed-up over a sequential execution on the fastest processor, that is over the schedule produced by SEQ.

6 Simulation Results

Here we present and interpret the results of the 1,296,000 simulation runs. As we have 15 runs for each combination of platform and DAG parameters, we consider the average of these results for a given {platform, DAG} couple.

In this study we only examine the evolution of our metrics depending on the variation of the platform parameters as we aim at quantifying the efficiency of our heuristics when heterogeneity is introduced, either by adding more clusters or by having different processor speeds.

Figure 2 shows the performance achieved by each of the considered heuristics on 1, 2, 4 and 8 clusters and the average makespan on all platforms. On top left we have a comparison on makespans. The top right part of the figure shows the compromise between completion time and peak resource utilization. On bottom left is another compromise between completion time and average resource utilization.

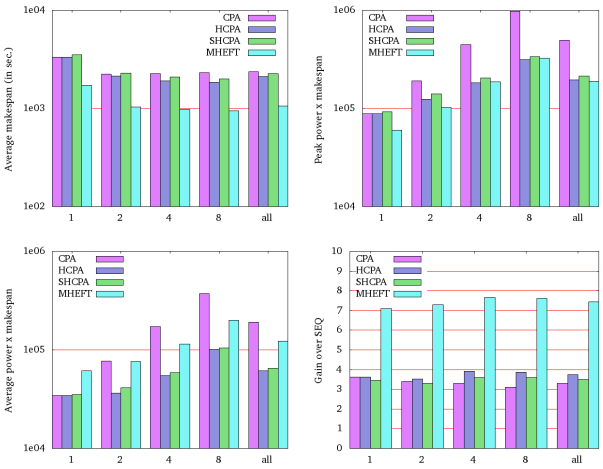

Figure 2. Impact of the number of clusters.

Let first focus on the rightest set of bars showing the comparison over the whole set of simulations. We can see that the two heuristics proposed in this paper achieve better makespan than CPA even if favored by a parallel task model without intra-task communications.

We can also notice that M-HEFT outperforms our heuristics in terms of completion time but at the price of an higher resource usage. This means that M-HEFT is efficient with respect to completion time but not to resource consumption. This is because the allocation process of M-HEFT is only focused on the reduction of the finish time of each task. As staying on the same set of processors takes less time than moving data, every costly task of a given path will be allocated on the same set of processors which is usually as large as a cluster. On the contrary our algorithms will allocate processors to tasks focusing on their impact over the completion time of the whole application. Only critical tasks are then placed on many processors.

We can conclude that M-HEFT and HCPA are complementary and using one or the other depends on the needs of the end user. If the user has a full access to several clusters with no restrictions, he/she should use M-HEFT to reduce the global completion time of the application. On the contrary, if he/she has access to a shared platform with accounting, a more rational resource usage becomes prevalent. HCPA will then guarantee a good tradeoff between performance and resource consumption. This is illustrated by Figure 3 that shows the comparison of the ratio between the makespans respectively achieved by M-HEFT and HCPA and the ratio between the average processing power usage of HCPA and M-HEFT. If M-HEFT is two times faster than HCPA over the range of experiments, the resource consumption of HCPA is 3.4 times better.

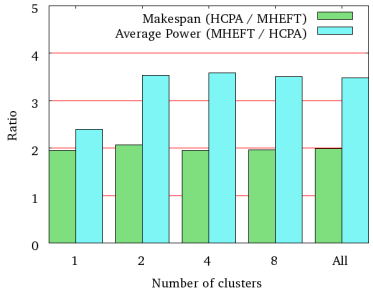

Figure 3. Comparison of makespan and resource consumption for M-HEFT and HCPA.

The four first sets of bars in Figure 2 show that on a single cluster CPA and HCPA produce similar schedules which it is not surprising as they share the same placement step. When the number of clusters increases, our algorithms lead to slightly better makespans. But if we look at the compromise between the makespan and the peak of resource usage induced by each heuristic, we can see that the resource consumption of CPA worsens with regard to that of our algorithms. Our algorithm gain is then not only on makespan but also on resource usage. This result is confirmed by the compromise between the makespan and the average resource usage. This can be explained by the fact that CPA can allocate processors of different clusters to a task. For some costly tasks, it may lead to a small gain on makespan but to a high increase of the resource consumption. We can finally see on the bottom right part of Figure 2, that the explicit handling of an heterogeneous platform leads to a performance increase. When we compare to an objective point of comparison such as a sequential schedule, we can see that the gap between the gain of CPA and that of HCPA increases.

Figure 4 shows the impact of the platform heterogeneity on the performance of the heuristics. As for platforms made of one cluster we can see that on homogeneous plat-

forms (in terms of computing power) CPA and HCPA still achieve the same makespan. The decrease of makespans with the increase of heterogeneity can be explained by the fact that with a high heterogeneity platform we have more fast processors that are used by the heuristics. When the heterogeneity increases HCPA also leads to slightly better makespans but produce schedules with a better resource usage than CPA. We can conclude that in the context for which CPA has been designed, HCPA produces the same schedules but efficiently extends it to heterogeneous platforms.

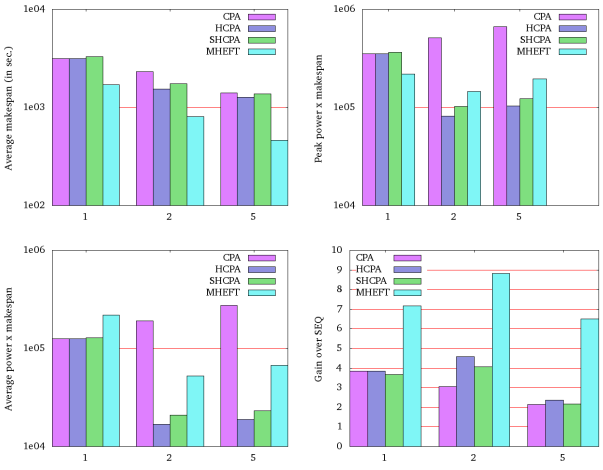

Figure 4. Impact of platform heterogeneity.

Finally if we compare the performance of HCPA with regard to that of S-HCPA, we can see that HCPA seems to be always better. But if we consider the simulation results independently, S-HCPA achieves the same or a better makespan for 30% of the simulations. This means that a list scheduling focused on the reduction of the critical path does not always minimizes the makespan of parallel applications and that other strategies have to be explored.

7 Conclusion and Future Work

Scheduling parallel tasks is a well-known technique for increasing the scalability of many parallel applications. In particular, it makes it possible for large-scale applications to efficiently exploit platforms that comprise several compute clusters. While parallel task scheduling algorithms have been proposed for homogeneous platforms, almost no work has been conducted in the case of heterogeneous platforms. In this paper we have addressed the need for efficient scheduling algorithms on heterogeneous platforms for applications made of parallel tasks. Our approach consists in adapting a efficient algorithm of the literature, CPA, in order to handle an heterogeneous platform. This is done by introducing the concept of *reference cluster* on which are made the processor allocations and that stands for the actual clusters of the platform. We developed two original heuristics, HCPA and S-HCPA with different placement steps. We evaluated the performance of our heuristics for a variety of platform scenarios and random DAGs. We compared HCPA and S-HCPA to the original CPA algorithm and to M-HEFT [1]. Our results showed that HCPA achieves better performance than CPA when platform are heterogeneous and we have discussed a number of trends apparent in our results. While this experimental study has value, our main contribution is a methodology to handle heterogeneity in parallel task scheduling algorithms designed for homogeneous platforms.

As a future work we aim at simulating more realistic platforms such as *Grid'5000* (`www.grid5000.fr`). Thus we can obtain more complex topologies with uncoupled subgraphs of clusters. We also plan to change our parallel task model to introduce intra-task communication costs either by using more realistic speedup based models [3] or by using some parallel task profiles depending on computation and communication complexities.

References

[1] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *10th Int. Euro-Par Conference*, volume 3149 of *LNCS*, pages 230–237, Pisa, Italy, Aug. 2004. Springer.

[2] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, Cancun, Mexico, 2000.

[3] A. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, 1997.

[4] P.-F. Dutot. Hierarchical Scheduling for Moldable Tasks. In *11th Int. Euro-Par Conference*, volume 3648 of *LNCS*, pages 302–311, Lisbon, Portugal, Aug. 2005. Springer.

[5] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, Tokyo, May 2003.

[6] R. Lepère, D. Trystram, and G. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *IJFCS*, 13(4):613–627, 2002.

[7] A. Radulescu, C. Nicolescu, A. van Gemund, and P. Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, Apr 2001.

[8] S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, Univ. of Illinois, Urbana-Champaign, 1996.

[9] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[10] J. Turek, J. Wolf, and P. Yu. Approximate Algorihtms for Scheduling Parallelizable Tasks. In *4th ACM Symp. on Parallel Algorithms and Architectures*, pages 323–332, 1992.